Documentation

***Blue Engagement***

Author: Andrei Lăuran

**1. Introduction**

The goal of this project was to develop a software solution that enhances the engagement between workers in the company named "Blue". The importance of employee engagement cannot be overstated, as it directly affects productivity, job satisfaction, and ultimately, the success of the company. With this in mind, I set out to create a software solution that not only improves engagement, but also provides a user-friendly experience for employees.

**2. Technology**

To effectively execute this assignment, I have meticulously chosen the .NET framework as the backend and have selected the SQLite database for development purposes. The reason being, SQLite is an incredibly useful database during the application development process as it eliminates the need to manage a separate database server or install any additional software. However, for production purposes, my preference would be PostgreSQL, due to its advanced features and robust capabilities. My choice stems from my confidence that this technology is at the forefront of current software development. For the frontend, I have opted for Angular as I believe that its development process closely aligns with that of .NET. To facilitate the development process, I have utilized Rider and WebStorm, products from JetBrains, renowned for their robust IntelliSense and user-friendly interface. These powerful tools have made the development process smoother, faster, and more efficient.

**3. Use Case Diagrams**

A use case diagram is a type of diagram that is used to represent interactions between a system and a user. It depicts the various ways in which the system can be used and the different types of users who interact with it. In short, a use case diagram defines what features the application has and the main actions the user can do.

**Diagram

Description automatically generatedUse Case:** Login

**Actors:** Employee, Manager

**Main success scenario:**

1. The user enters their email address or username and password

2. If the credentials are ok, create a token and return a user dto containing that token.

**Use Case:** Register

**Actors:** Employee

**Main success scenario:**

1. The user enters the required details

2. If the email or username is not already in use, creates the token and returns a user dto containing the token (this will automatically log the user in after register)

**Use Case:** Get All Badges

**Actors:** Employee, Manager

**Main success scenario:**

1. A list containing all the badges will be returned from the db.

**Use Case:** Create Badge

**Actors:** Manager

**Main success scenario:**

1. The manager enters the badge description and the picture URL of the badge image

2. The badge is added in the database

**Use Case:** Delete Badge

**Actors:** Manager

**Main success scenario:**

1. After manager has selected the badge, it will be deleted from the db.

**Use Case:** Get Own Quests

**Actors:** Employee, Manager

**Main success scenario:**

1. A list containing the quests proposed by the logged in user will be retrieved from the db

**Use Case:** Get other’s Quests

**Actors:** Employee, Manager

**Main success scenario:**

1. The user can see all the proposed quests on the ‘Quests’ page, and, by pressing a button, the details of a quest, including even the quest status of the logged in user (Not accepted, Accepted or Completed) alongside with all quests proposed by the same author will be retrieved.

**Use Case:** Accept a quest

**Actors:** Employee, Manager

**Main success scenario:**

1. The user enters the page containing the quest details

2. By pressing a button, the server checks if the user hasn’t already accepted or completed the quest and creates a taken quest, having status ‘Accepted’

**Use Case:** Create Quest

**Actors:** Employee, Manager

**Main success scenario:**

1. The user enters the required details

2. If the user has enough points, the points needed to create the quest will be deducted from the user and the quest will be created and returned.

**Use Case:** Delete Own Quest

**Actors:** Employee, Manager

**Main success scenario:**

1. The user selects the quest he wants to delete.

2. The server will validate the user’s request (will check if either the logged in user is the quest’s author, either if he’s a manager) and then will delete the quest from the db.

**Use Case:** See Leaderboard

**Actors:** Employee, Manager

**Main success scenario:**

1. The logged in user makes a request and the list of users sorted descending by their score will be returned from the database. For the demo purpose, all the users will be retrieved. In production, only let’s say 100 users should be retrieved.

**Use Case:** Get All Users

**Actors:** Manager

**Main success scenario:**

1. A list containing all users will be retrieved. At first, I thought of it as a method to check the users, or even remove (ban) a user if the user doesn’t behave or is fired or etc.

**4. Architecture**

For this project, for the backend, I have decided to go with the “Clean Architecture” because Clean Architecture style focus on a loosely coupled implementation of use cases. Use cases as central organizing structure, decoupled from frameworks and technology details.

**Diagram

Description automatically generated**

**Core Layer:**

The “Core” Layer should not depend on any other layer, so the dependencies are inverted. This is achieved by adding here the interfaces that are implemented in the other layers as well as the entities, dtos, enums, constants etc.

**Application Layer:**

Application Layer (API in our case) implements the use cases of the application based on the domain. In this layer are the controllers, which are responsible for handling the user’s request and call the required method for the user’s desired action.

**Infrastructure Layer:**

This layer is responsible to implement the Contracts (Interfaces/Adapters) defined within the application layer to the Secondary Actors. Infrastructure Layer supports other layer by implementing the abstractions and integrations to 3rd-party library and systems. Here is where the repositories and services are defined and implemented.

With regards to the frontend architecture, I have adhered to the Angular team's recommendations by utilizing a core folder, a shared folder, and a few feature folders or modules.

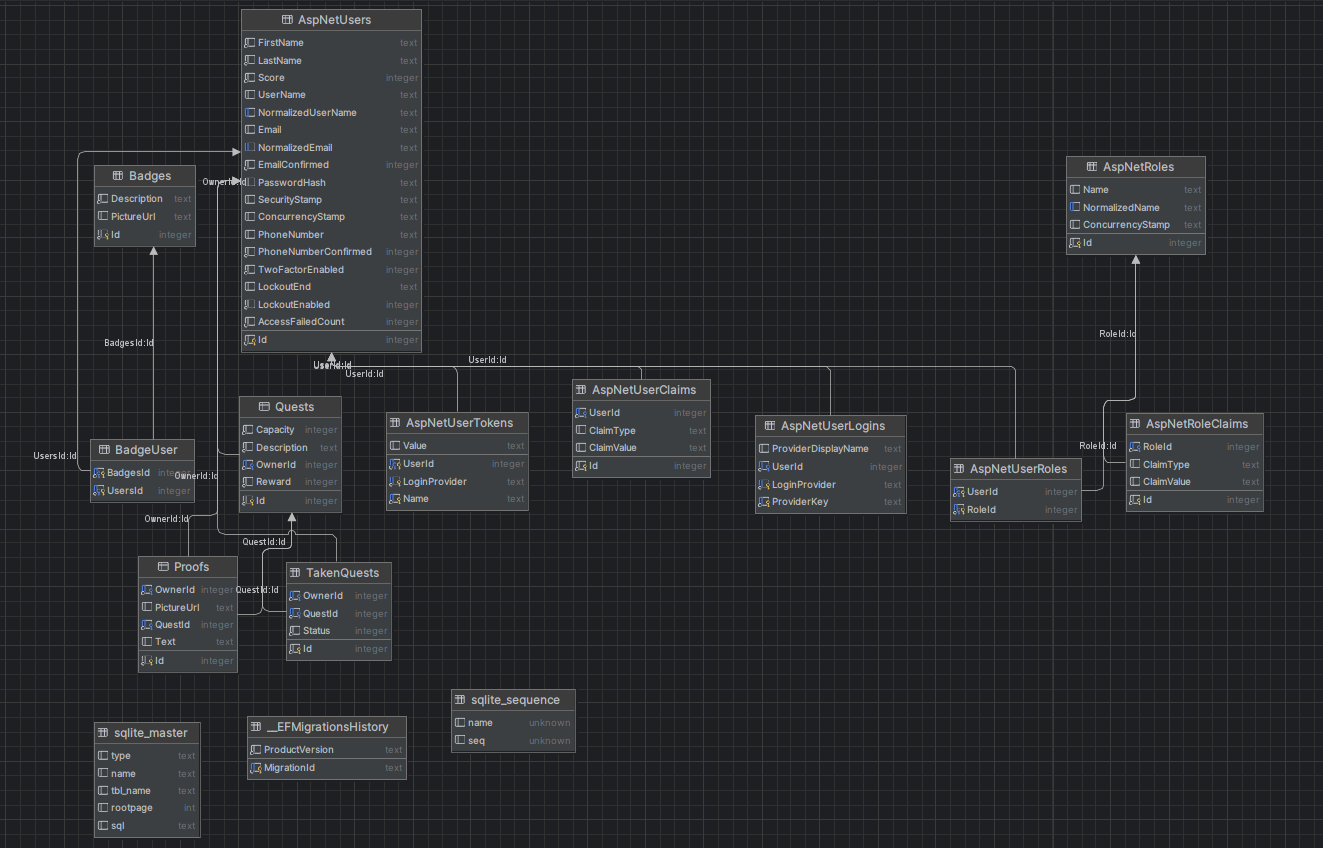
Text

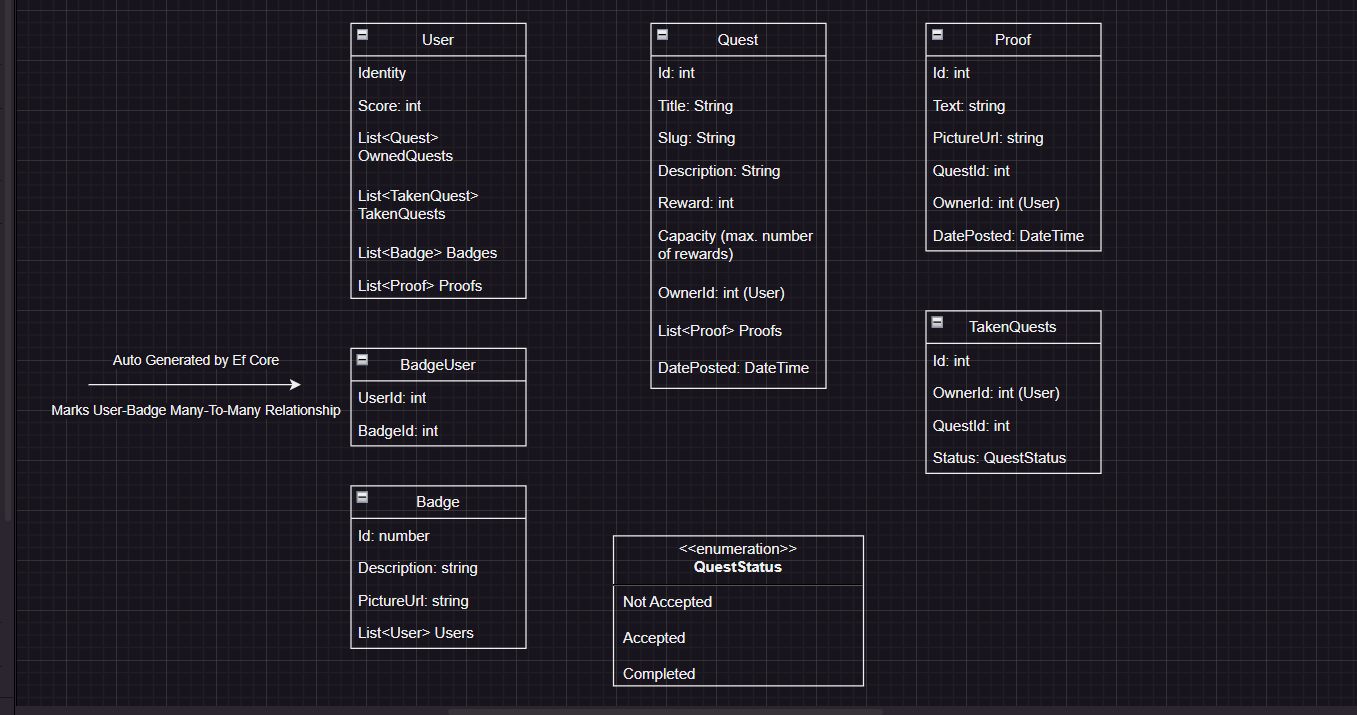
Description automatically generated**5. Package Diagram**

A package diagram is a UML (Unified Modeling Language) diagram that illustrates the organization and dependencies of software packages and their components. A package is a collection of related classes, interfaces, and other packages that provide a cohesive set of features and functionality within a software system.

In a package diagram, packages are represented as rectangles, and their contents are shown as smaller rectangles inside them. Arrows between packages indicate the dependencies between them, with the arrow pointing from the dependent package to the package it depends on. These dependencies can be used to visualize the structure of a software system, identify potential design issues, and help with maintenance and refactoring.

Package diagrams can also be used to represent different levels of abstraction within a software system, such as high-level system components or low-level modules within a package. They are useful for both developers and stakeholders, as they provide a visual representation of the software architecture that can be easily understood and communicated.

**6. Database Diagram**

This is the final database diagram generated by the editor. Some tables were added automatically by the identity since I made use of it for the log in / register part. I will also add below the sketch for the models (from which the database was generated), sketch which was made manually by myself for developing the project, which is much simpler, and, in my opinion, more explicative.

**7. Endpoints Documentation**

For the endpoint documentation I have added swagger in my project. Swagger can be found at <https://localhost:5001/swagger/index.html>. However, since this application requires authentication in order to function, I strongly recommend putting these into test using postman, since the authorization part will be needed.

How to test with postman?

Add to header the following key-value pair:

Key: Authorization

Value: Bearer <login\_token>

Login token will be received as a response either when the user logs in, either when he authenticates.

Additionally, I will be making my Postman workspace publicly available, enabling you to effortlessly and expeditiously test all the functionalities currently present in the application:

**https://www.postman.com/material-participant-85315924**